Kapitel 8: Datenaufbereitung - Verknüpfen und Umformen

McKinney, W. (2017). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. 2. Auflage. Sebastopol, CA [u. a.]: O'Reilly.

Überarbeitet: armin.baenziger@zhaw.ch, 28. Februar 2020

- In vielen Anwendungen können Daten über mehrere Dateien oder Datenbanken verteilt sein oder in einer Form angeordnet sein, die nicht leicht zu analysieren ist.
- Dieses Kapitel konzentriert sich auf Tools zum Kombinieren, Verknüpfen und Umformen von Daten.
- Zuerst führen wir das Konzept der hierarchischen Indexierung in Pandas ein, das in einigen dieser Operationen ausführlich verwendet wird.
- Danach betrachten wir spezielle Datenmanipulationen. Verschiedene Anwendungen dieser Werkzeuge werden in Kapitel 14 (bzw. im Unterricht) vorgestellt, wo konkrete (grössere) Datensätze analysiert werden.

In [1]:
```
%autosave 0
```

```
Autosave disabled
```

In [2]:
```
# Bibliotheken mit üblichen Abkürzungen laden:
import numpy as np
import pandas as pd
```

Hierarchische Indexierung

- Die hierarchische Indexierung ist ein Merkmal von Pandas, mit dem man mehrere (zwei oder mehr) Indexstufen auf einer Achse haben kann.
- Etwas abstrakt bietet es eine Möglichkeit, mit *höherdimensionalen Daten* (z. B. drei Dimensionen) in einer niederdimensionalen Form (z. B. zwei Dimensionen; Zeilen und Spalten) zu arbeiten.
- Wir beginnen mit einem einfachen Beispiel: Einer Series mit einer *Liste von Listen (oder Arrays)* als Index:

In [3]:
```
# Beispieldaten generieren:
Jahr    = [2018]*4 + [2019]*4 + [2020]*2
Quartal = [1, 2, 3, 4]*2 + [1, 2]

Umsatz = pd.Series([11, 13, 10, 19, 10, 12, 9, 21, 13, 15],
          index = [Jahr, Quartal])*10

Umsatz.index.names = ['Jahr', 'Quartal']

Umsatz
```

Out[3]:
```
Jahr  Quartal
2018  1          110
      2          130
      3          100
      4          190
2019  1          100
      2          120
      3           90
      4          210
2020  1          130
      2          150
dtype: int64
```

Beachten Sie den hierarchischen Index: Hierarchieebene 1 des Index sind die Jahre und Hierarchieebene 2 die Quartale.

Bei einem hierarchisch indizierten Objekt ist eine so genannte *partielle Indizierung* möglich, mit der Teilmengen der Daten übersichtlich ausgewählt werden können:

In [4]:
```
# Alle Umsätze im 2019 (erste Hierarchieebene):
Umsatz.loc[2019]
```

Out[4]:
```
Quartal
1    100
2    120
3     90
4    210
dtype: int64
```

In [5]:
```
# Alle Umsätze ab 2019:
Umsatz.loc[2019:]
```

Out[5]:
```
Jahr  Quartal
2019  1          100
      2          120
      3           90
      4          210
2020  1          130
      2          150
dtype: int64
```

In [6]:
```
# Alle Umsätze von 2018 und 2020:
Umsatz.loc[[2018, 2020]]
```

Out[6]:
```
Jahr  Quartal
2018  1          110
      2          130
      3          100
      4          190
2020  1          130
      2          150
dtype: int64
```

Es ist auch möglich, von einer "inneren" Ebene zu selektieren (hier 1. Quartal):

In [7]:
```
Umsatz.loc[:, 1]  # Alle Umsätze im 1. Quartal
```

Out[7]:
```
Jahr
2018    110
2019    100
2020    130
dtype: int64
```

Kontrollfragen:

In [8]:
```
# Gegeben:
Umsatz
```

Out[8]:
```
Jahr  Quartal
2018  1          110
      2          130
      3          100
      4          190
2019  1          100
      2          120
      3           90
      4          210
2020  1          130
      2          150
dtype: int64
```

In [9]:
```
# Frage 1: Was ist der Output?
Umsatz.loc[2019].sum()
```

Out[9]:
```
520
```

In [10]:
```
# Frage 2: Was ist der Output?
Umsatz.loc[:, 2].sum()
```

Out[10]:
```
400
```

- Die *hierarchische Indizierung* spielt eine wichtige Rolle bei der Umformung von Daten und gruppenbasierten Vorgängen wie der Bildung von *Pivot-Tabellen*.
- Beispielsweise könnten die Daten mithilfe ihrer `unstack` -Methode in ein DataFrame umgeformt werden:

In [11]:
```
# Zur Erinnerung:
Umsatz   # eine Series mit hierarchischem Index
```

Out[11]:
```
Jahr  Quartal
2018  1          110
      2          130
      3          100
      4          190
2019  1          100
      2          120
      3           90
      4          210
2020  1          130
      2          150
dtype: int64
```

In [12]:
```
df = Umsatz.unstack()
# Die zweite (innere) Hierarchieebene wird in die Spalten
# "gedreht" (vom "Long"- ins "Wide"-Format).
df
```

Out[12]:

Quartal	1	2	3	4
Jahr				
2018	110.0	130.0	100.0	190.0
2019	100.0	120.0	90.0	210.0
2020	130.0	150.0	NaN	NaN

Es entstehen zwei `NaN` am Ende, da die entsprechenden Quartalszahlen im letzten Jahr fehlen.

Es wäre auch möglich gewesen, die erste Hierarchieebene in die Spalten zu drehen:

In [13]:
```
Umsatz.unstack(level=0)
```

Out[13]:

Jahr	2018	2019	2020
Quartal			
1	110.0	100.0	130.0
2	130.0	120.0	150.0
3	100.0	90.0	NaN
4	190.0	210.0	NaN

Die inverse Operation ist `stack` :

In [14]:
```
df.stack()  # Stack "dreht" vom "Wide"- ins "Long"-Format
```

Out[14]:
```
Jahr  Quartal
2018  1          110.0
      2          130.0
      3          100.0
      4          190.0
2019  1          100.0
      2          120.0
      3           90.0
      4          210.0
2020  1          130.0
      2          150.0
dtype: float64
```

Bei einem DataFrame kann *jede Achse* (Zeilen oder Spalten) einen hierarchischen Index haben:

In [15]:
```
df
```

Out[15]:

Quartal	1	2	3	4
Jahr				
2018	110.0	130.0	100.0	190.0
2019	100.0	120.0	90.0	210.0
2020	130.0	150.0	NaN	NaN

In [16]:
```
df.columns = [['Halbjahr_1']*2 + ['Halbjahr_2']*2,
              [1, 2, 3, 4]]
df
```

Out[16]:

	Halbjahr_1		Halbjahr_2	
	1	2	3	4
Jahr				
2018	110.0	130.0	100.0	190.0
2019	100.0	120.0	90.0	210.0
2020	130.0	150.0	NaN	NaN

In [17]:
```
df.Halbjahr_2  # nur Spalten mit Überschrift "Halbjahr_2" auswählen
```

Out[17]:

Jahr	3	4
2018	100.0	190.0
2019	90.0	210.0
2020	NaN	NaN

Kontrollfrage:

In [18]:
```
# Gegeben:
ser = pd.Series([1, 3, 2, 3, 5],
                index=[list('AABBB'), list('ababc')])
ser
```

Out[18]:
```
A  a    1
   b    3
B  a    2
   b    3
   c    5
dtype: int64
```

In [19]:
```
# Frage: Was ist der Output?
ser.unstack()
```

Out[19]:

	a	b	c
A	1.0	3.0	NaN
B	2.0	3.0	5.0

Zusammenfassende Statistiken nach Hierarchieebene

Viele beschreibende und zusammenfassende Statistiken zu DataFrames und Series verfügen über eine **`level` -Option**, *mit der man die Hierarchieebene angeben kann, nach der auf einer bestimmten Achse aggregiert werden soll*.

In [20]:
```
# Beispieldaten:
Umsatz_df = pd.DataFrame({'Filiale_A': [11, 13, 10, 19, 10, 12, 9, 21, 13, 15],
                          'Filiale_B': [9, 11, 8, 15, 10, 11, 10, 16, 11, 13]},
                          index = [Jahr, Quartal]) * 10

Umsatz_df.index.names = ['Jahr', 'Quartal']

Umsatz_df
```

Out[20]:

		Filiale_A	Filiale_B
Jahr	**Quartal**		
2018	**1**	110	90
	2	130	110
	3	100	80
	4	190	150
2019	**1**	100	100
	2	120	110
	3	90	100
	4	210	160
2020	**1**	130	110
	2	150	130

In [21]:
```
Umsatz_df.sum()    # Spaltensummen
```

Out[21]:
```
Filiale_A    1330
Filiale_B    1140
dtype: int64
```

In [22]:
```
Umsatz_df.sum(level='Quartal')    # Spaltensummen nach 'Quartal'
```

Out[22]:

	Filiale_A	Filiale_B
Quartal		
1	340	300
2	400	350
3	190	180
4	400	310

Kontrollfragen:

In [23]:
```
# Gegeben:
Umsatz_df
```

Out[23]:

		Filiale_A	Filiale_B
Jahr	**Quartal**		
2018	**1**	110	90
	2	130	110
	3	100	80
	4	190	150
2019	**1**	100	100
	2	120	110
	3	90	100
	4	210	160
2020	**1**	130	110
	2	150	130

In [24]:
```
# Frage 1: Was ist der Output?
Umsatz_df.sum(level='Jahr')
```

Out[24]:

	Filiale_A	Filiale_B
Jahr		
2018	530	430
2019	520	470
2020	280	240

In [25]:
```
# Frage 2: Was ist der Output?
Umsatz_df['Filiale_B'].mean(level='Jahr')
```

Out[25]:
```
Jahr
2018    107.5
2019    117.5
2020    120.0
Name: Filiale_B, dtype: float64
```

Indexierung mit einer Spalte eines DataFrame

- Es ist nicht ungewöhnlich, dass man eine oder mehrere Spalten aus einem DataFrame als (Zeilen-) Index verwenden möchte.
- Alternativ ist es möglich, den Zeilenindex in die Spalten des DataFrames zu verschieben.
- Beispiel:

In [26]:
```
# Beispieldaten:
Umsatz_Daten = Umsatz_df.reset_index()
Umsatz_Daten
```

Out[26]:

	Jahr	Quartal	Filiale_A	Filiale_B
0	2018	1	110	90
1	2018	2	130	110
2	2018	3	100	80
3	2018	4	190	150
4	2019	1	100	100
5	2019	2	120	110
6	2019	3	90	100
7	2019	4	210	160
8	2020	1	130	110
9	2020	2	150	130

In [27]:
```
# Spalten "Jahr" und "Quartal" in den Index verschieben:
Umsatz_Daten.set_index(['Jahr', 'Quartal'], inplace=True)
Umsatz_Daten
```

Out[27]:

		Filiale_A	Filiale_B
Jahr	**Quartal**		
2018	**1**	110	90
	2	130	110
	3	100	80
	4	190	150
2019	**1**	100	100
	2	120	110
	3	90	100
	4	210	160
2020	**1**	130	110
	2	150	130

Kontrollfragen:

In [28]:
```
# Gegeben:
drinks = pd.read_csv('../weitere_Daten/drinksbycountry.csv', usecols=[0, 4, 5])
drinks.rename(columns={'total_litres_of_pure_alcohol': 'alcohol'}, inplace=True)
drinks5 = drinks.sample(5, random_state=6).reset_index(drop=True)
drinks5
```

Out[28]:

	country	alcohol	continent
0	Eritrea	0.5	Africa
1	Angola	5.9	Africa
2	Switzerland	10.2	Europe
3	Bangladesh	0.0	Asia
4	Kuwait	0.0	Asia

In [29]:
```
# Frage 1: Was ist der Output?
drinks5HI = drinks5.set_index(['continent', 'country']).sort_index()
drinks5HI
```

Out[29]:

		alcohol
continent	**country**	
Africa	**Angola**	5.9
	Eritrea	0.5
Asia	**Bangladesh**	0.0
	Kuwait	0.0
Europe	**Switzerland**	10.2

In [30]:
```
# Frage 2: Was ist der Output?
drinks5HI.loc['Asia']
```

Out[30]:

	alcohol
country	
Bangladesh	0.0
Kuwait	0.0

In [31]:
```
# Frage 3: Was ist der Output?
drinks5HI.loc[('Asia', 'Kuwait')]
```

Out[31]:
```
alcohol    0.0
Name: (Asia, Kuwait), dtype: float64
```

Kombinieren und Verknüpfen von Datensätzen

Wir folgen in diesem Abschnitt dem **"Cheat Sheet"** *"Data Wrangling with pandas"* statt dem Lehrmittel, das für den Einstieg zu viele Funktionalitäten bespricht, wie ich meine. Hier versuche ich Ihnen einen guten Überblick über die Möglichkeiten zu geben und bei konkreten Problemstellungen kann man sich dann mit den Details beschäftigen. Es werden drei Funktionen/Methoden besprochen, nämlich `concat`, `merge` und `join`. Was unterscheidet diese *grundsätzlich*?

- Mit **`concat`** werden DataFrames aneinandergereiht ("gestapelt"), entweder untereinander (`axis=0`) oder nebeneinander (`axis=1`).
- Eine weitere Möglichkeit, DataFrames zu kombinieren, besteht darin, *in jedem Dataset Spalten zu verwenden, die gemeinsame Werte enthalten (eine allgemeine eindeutige ID)*. Hierzu verwenden wir grundsätzlich die Funktion **`merge`**.
  - Die Kombination von DataFrames mit einem gemeinsamen Feld heisst *"Joining"*.
  - Die Spalten mit den gemeinsamen Werten heissen "Join Key(s)".
  - Das Verbinden von DataFrames auf diese Weise ist oft nützlich, wenn ein DataFrame eine "Nachschlagetabelle" ist, die zusätzliche Daten enthält, die wir in die andere einschliessen möchten (*"many-to-one merges"*).
  - Dieser Prozess zum Verknüpfen von Tabellen ähnelt dem, den wir mit Tabellen in SQL-Datenbanken ausführen.
- *Falls die gemeinsamen Werte in den Indizes vorliegen*, kann man statt `merge` (mit den Argumenten `left_index=True`, `right_index=True`) die Methode **`join`** verwenden.

Aneinanderreihen: `concat`

Mit `concat` werden einfach die Zeilen (oder Spalten, falls `axis=1`) von DataFrames aneinandergereiht bzw. gestapelt. *Dabei können Duplikate entstehen*.

In [32]:
```
# Beispieldaten generieren:
umsatz_df1 = Umsatz_df.loc[:2019]
umsatz_df1
```

Out[32]:

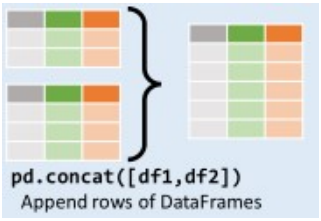
		Filiale_A	Filiale_B
Jahr	**Quartal**		
2018	**1**	110	90
	2	130	110
	3	100	80
	4	190	150
2019	**1**	100	100
	2	120	110
	3	90	100
	4	210	160

In [33]:
```
# Weitere Beispieldaten generieren:
umsatz_df2 = Umsatz_df.loc[2020:]
umsatz_df2
```

Out[33]:

		Filiale_A	Filiale_B
Jahr	**Quartal**		
2020	**1**	130	110
	2	150	130

Zeilen anhängen:



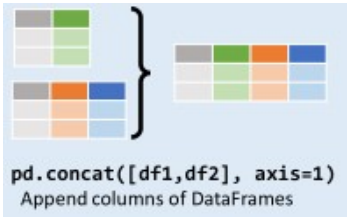
In [34]:
```
pd.concat([umsatz_df1, umsatz_df2])
```

Out[34]:

		Filiale_A	Filiale_B
Jahr	**Quartal**		
2018	**1**	110	90
	2	130	110
	3	100	80
	4	190	150
2019	**1**	100	100
	2	120	110
	3	90	100
	4	210	160
2020	**1**	130	110
	2	150	130

Man beachte, dass eine *Liste* mit DataFrames übergeben werden muss.

Spalten anhängen:



In [35]:
```
# Beispieldaten generieren:
Umsatz_df2 = Umsatz_df.copy()
Umsatz_df2.rename({'Filiale_A': 'Filiale_C',
                   'Filiale_B': 'Filiale_D'},
                   axis=1, inplace=True)
Umsatz_df2
```

Out[35]:

		Filiale_C	Filiale_D
Jahr	**Quartal**		
2018	**1**	110	90
	2	130	110
	3	100	80
	4	190	150
2019	**1**	100	100
	2	120	110
	3	90	100
	4	210	160
2020	**1**	130	110
	2	150	130

In [36]:
```
pd.concat([Umsatz_df, Umsatz_df2], axis=1)
```

Out[36]:

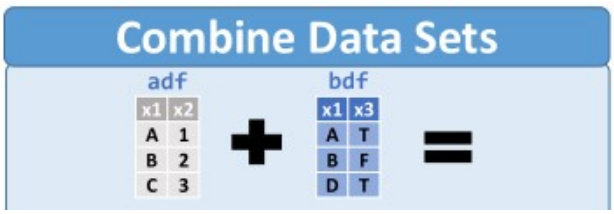
		Filiale_A	Filiale_B	Filiale_C	Filiale_D
Jahr	**Quartal**				
2018	**1**	110	90	110	90
	2	130	110	130	110
	3	100	80	100	80
	4	190	150	190	150
2019	**1**	100	100	100	100
	2	120	110	120	110
	3	90	100	90	100
	4	210	160	210	160
2020	**1**	130	110	130	110
	2	150	130	150	130

Spalten werden flexibler und sicherer (korrekt ausgerichtet) mit `merge` oder `join` einem DataFrame hinzugefügt. Damit befasst sich der nächste Abschnitt.

DataFrame im Datenbankstil verknüpfen: `merge` und `join`

- *Merge-* oder *join*-Operationen kombinieren Datasets, *indem sie Zeilen mit einem oder mehreren Schlüsseln verknüpfen*.
- Diese Operationen sind für *relationale Datenbanken* (z.B. SQL-basiert) von zentraler Bedeutung.
- Die `merge` -Funktion in Pandas ist der Haupteinstiegspunkt für die Verwendung dieser Algorithmen.

Verknüpfen: `merge`



In [37]:
```
# Beispieldaten erzeugen:
adf = pd.DataFrame({'x1': ['A', 'B', 'C'], 'x2': [1, 2, 3]})
bdf = pd.DataFrame({'x1': ['A', 'B', 'D'], 'x3': ['T', 'F', 'T']})
adf
```

Out[37]:

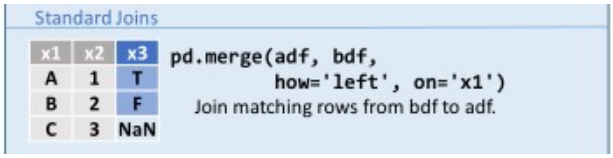
	x1	x2
0	A	1
1	B	2
2	C	3

In [38]:
```
bdf
```

Out[38]:

	x1	x3
0	A	T
1	B	F
2	D	T

Zuerst mergen wir die DataFrames `adf` und `bdf` über die Spalte/Variable `x1` . Mit dem Argument `how=left` werden die Daten an der Variable (hier `x1`) des "linken" (ersten) DataFrame ausgerichtet. Werte der Merge-Variablen (`x1`), die *nur* im rechten (zweiten) DataFrame vorkommen, werden nicht übernommen.



In [39]:
```
pd.merge(adf, bdf, how='left', on='x1')
```

Out[39]:

	x1	x2	x3
0	A	1	T
1	B	2	F
2	C	3	NaN

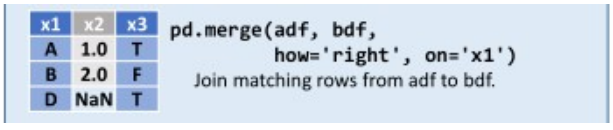
In [40]:
```
# Oder als Methode mit gleichem Ergebnis:
adf.merge(bdf, how='left', on='x1')
```

Out[40]:

	x1	x2	x3
0	A	1	T
1	B	2	F
2	C	3	NaN

C in Merge-Spalte `x1` gibt es nur im ersten (linken) DataFrame, so dass die Variable `x3` an der Stelle ein `NaN` enthält.
D in Merge-Spalte `x1` gibt es nur im zweiten (rechten) DataFrame und wird bei `how='left'` nicht übernommen.

Man kann auch an der Merge-Variable (`x1`) des rechten (zweiten) DataFrame ausrichten.

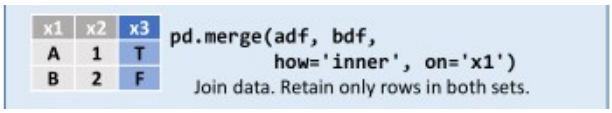


In [41]:
```
pd.merge(adf, bdf, how='right', on='x1')
```

Out[41]:

	x1	x2	x3
0	A	1.0	T
1	B	2.0	F
2	D	NaN	T

Mit dem Argument `how=inner` werden nur Zeilen über die Merge-Variable verbunden, bei denen die Ausprägungen der Merge-Variable in beiden DataFrames vorkommen (Schnittmenge).

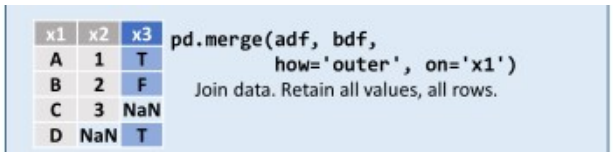


In [42]:
```
pd.merge(adf, bdf, how='inner', on='x1')
```

Out[42]:

	x1	x2	x3
0	A	1	T
1	B	2	F

Mit dem Argument `how=outer` werden Zeilen über die Merge-Variable verbunden, bei denen die Ausprägungen der Merge-Variable in einem der beiden DataFrames vorkommen (Vereinigungsmenge).



In [43]:
```
pd.merge(adf, bdf, how='outer', on='x1')
```

Out[43]:

	x1	x2	x3
0	A	1.0	T
1	B	2.0	F
2	C	3.0	NaN
3	D	NaN	T

Many-to-one Merges: Das Verbinden von DataFrames ist oft nützlich, wenn ein DataFrame eine "Nachschlagetabelle" ist, die zusätzliche Daten enthält, die wir in die andere einschliessen möchten (*"many-to-one merges"*).

In [44]:
```
Stammdaten = pd.DataFrame({'Person': list('ABCD'),
                           'Arbeitgeber': ['ZHAW', 'UBS', 'UBS', 'ZHAW']},
                           columns=['Person', 'Arbeitgeber'])
Stammdaten
```

Out[44]:

	Person	Arbeitgeber
0	A	ZHAW
1	B	UBS
2	C	UBS
3	D	ZHAW

In [45]:
```
Nachschlagetabelle = pd.DataFrame({'Arbeitgeber': ['UBS', 'ZHAW'],
                                   'Trägerschaft': ['privat', 'öffentlich'],
                                   'Branche': ['Bank', 'Hochschule']})
Nachschlagetabelle
```

Out[45]:

	Arbeitgeber	Trägerschaft	Branche
0	UBS	privat	Bank
1	ZHAW	öffentlich	Hochschule

In [46]:
```
Stammdaten.merge(Nachschlagetabelle, on='Arbeitgeber')
```

Out[46]:

	Person	Arbeitgeber	Trägerschaft	Branche
0	A	ZHAW	öffentlich	Hochschule
1	D	ZHAW	öffentlich	Hochschule
2	B	UBS	privat	Bank
3	C	UBS	privat	Bank

Kontrollfragen:

In [47]:
```
# Gegeben:
df1 = pd.DataFrame({'x': ['A', 'B', 'C'],
                    'y': [1, 2, 3]})
df1
```

Out[47]:

	x	y
0	A	1
1	B	2
2	C	3

In [48]:
```
# Gegeben ein weiteres DataFrame:
df2 = pd.DataFrame({'x': ['A', 'A', 'B'],
                    'z': [11, 12, 13]})
df2
```

Out[48]:

	x	z
0	A	11
1	A	12
2	B	13

In [49]:
```
# Frage 1: Was ist der Output?
pd.merge(df1, df2, on='x', how='outer')
```

Out[49]:

	x	y	z
0	A	1	11.0
1	A	1	12.0
2	B	2	13.0
3	C	3	NaN

In [50]:
```
# Frage 2: Was ist der Output?
pd.merge(df1, df2, on='x', how='right')
```

Out[50]:

	x	y	z
0	A	1	11
1	A	1	12
2	B	2	13

Verknüpfen am Index: `join`

Oft ist die Information, die zum Verbinden zweier DataFrames nötig ist, im Index der DataFrames enthalten.

In [51]:
```
# Beispieldaten erzeugen:
adf.set_index('x1', inplace=True)
adf
```

Out[51]:

	x2
x1	
A	1
B	2
C	3

In [52]:
```
bdf.set_index('x1', inplace=True)
bdf
```

Out[52]:

	x3
x1	
A	T
B	F
D	T

Mit `merge()` und den Argumenten `left_index=True` und `right_index=True` können die zwei DataFrames verbunden werden.

In [53]:
```
adf.merge(bdf, left_index=True, right_index=True, how='inner')
```

Out[53]:

	x2	x3
x1		
A	1	T
B	2	F

Einfacher geht es mit der `join` -Methode.

In [54]:
```
adf.join(bdf, how='inner')
```

Out[54]:

x1	x2	x3
A	1	T
B	2	F

Kontrollfrage:

In [55]:
```
# Gegeben:
Umsatz_df
```

Out[55]:

Jahr	Quartal	Filiale_A	Filiale_B
2018	**1**	110	90
	2	130	110
	3	100	80
	4	190	150
2019	**1**	100	100
	2	120	110
	3	90	100
	4	210	160
2020	**1**	130	110
	2	150	130

In [56]:
```
# Gegeben:
Umsatz_df2
```

Out[56]:

Jahr	Quartal	Filiale_C	Filiale_D
2018	**1**	110	90
	2	130	110
	3	100	80
	4	190	150
2019	**1**	100	100
	2	120	110
	3	90	100
	4	210	160
2020	**1**	130	110
	2	150	130

In [57]:
```
# Frage: Was ist der Output?
Umsatz_df.join(Umsatz_df2)
```

Out[57]:

		Filiale_A	Filiale_B	Filiale_C	Filiale_D
Jahr	**Quartal**				
2018	**1**	110	90	110	90
	2	130	110	130	110
	3	100	80	100	80
	4	190	150	190	150
2019	**1**	100	100	100	100
	2	120	110	120	110
	3	90	100	90	100
	4	210	160	210	160
2020	**1**	130	110	130	110
	2	150	130	150	130

- Wir haben damit die wichtigsten Funktionalitäten bezüglich Kombinieren von Daten-Sets *im Grundsatz* besprochen.
- Das Lehrmittel vertieft die Ausführungen und bespricht insbesondere auch:
  - Many-to-many Merges
  - Merge bei hierarchischen Indizes

Rekapitulation: Was ist nochmals der Unterschied zwischen `merge`, `join` und `concat`?

- `concat` wird verwendet, um ein (oder mehrere) DataFrame(s) an ein anderes untereinander anzuhängen (oder auch seitwärts, falls `axis=1` gesetzt ist).
- Ein wesentlicher Unterschied zu `concat` besteht darin, dass `merge` verwendet wird, um zwei (oder mehr) DataFrames auf der *Basis von Werten gemeinsamer Spalten* zu verbinden (Indizes können auch verwendet werden, `left_index=True` und/oder `right_index=True`).
- `join` kann verwendet werden, um zwei DataFrames *auf der Basis des Index* zusammenzuführen (was einfacher ist, als `merge` mit der Option `left_index=True` und/oder `right_index=True` zu verwenden).

Umformen und Transponieren (Reshaping and Pivoting)

Umformen bei hierarchischem Index

- Die hierarchische Indexierung bietet eine konsistente Möglichkeit, Daten in einem DataFrame neu anzuordnen.
- Es gibt zwei Hauptaktionen (welche oben bereits kurz eingeführt wurden):
  - **`stack`**: Rotiert oder schwenkt von den Spalten in den Daten zu den Zeilen
  - **`unstack`**: Rotiert von den Zeilen in die Spalten

In [58]:
```
# Beispieldaten:
Umsatz_df
```

Out[58]:

		Filiale_A	Filiale_B
Jahr	**Quartal**		
2018	**1**	110	90
	2	130	110
	3	100	80
	4	190	150
2019	**1**	100	100
	2	120	110
	3	90	100
	4	210	160
2020	**1**	130	110
	2	150	130

In [59]:
```
Long_format = Umsatz_df.stack()
Long_format
```

Out[59]:
```
Jahr  Quartal
2018  1        Filiale_A    110
               Filiale_B     90
      2        Filiale_A    130
               Filiale_B    110
      3        Filiale_A    100
               Filiale_B     80
      4        Filiale_A    190
               Filiale_B    150
2019  1        Filiale_A    100
               Filiale_B    100
      2        Filiale_A    120
               Filiale_B    110
      3        Filiale_A     90
               Filiale_B    100
      4        Filiale_A    210
               Filiale_B    160
2020  1        Filiale_A    130
               Filiale_B    110
      2        Filiale_A    150
               Filiale_B    130
dtype: int64
```

In [60]:
```
Wide_format = Long_format.unstack()
Wide_format
```

Out[60]:

		Filiale_A	Filiale_B
Jahr	**Quartal**		
2018	**1**	110	90
	2	130	110
	3	100	80
	4	190	150
2019	**1**	100	100
	2	120	110
	3	90	100
	4	210	160
2020	**1**	130	110
	2	150	130

- Standardmässig wird die innerste Ebene "entstapelt" (dasselbe gilt für `stack`).
- Sie können eine andere Ebene entstapeln, indem Sie eine Ebenennummer oder Namen übergeben:

Pivoting vom "long"- zum "wide"-Format

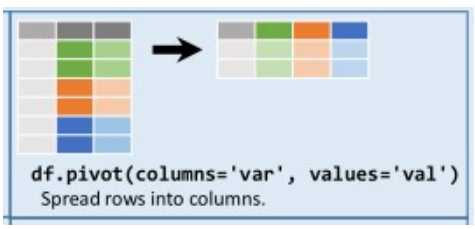
- Eine gängige Möglichkeit, mehrere Zeitreihen in Datenbanken (z. B. MySQL) und CSV-Dateien zu speichern, ist das so genannte "Long"- oder "Stacked"-Format.
- Beispiel:

In [61]:
```
long_data = pd.read_csv('../weitere_Daten/long_data.csv')
long_data.head()
```

Out[61]:

	date	variable	value
0	1959-03-31	realgdp	2710.349
1	1959-03-31	infl	0.000
2	1959-03-31	unemp	5.800
3	1959-06-30	realgdp	2778.801
4	1959-06-30	infl	2.340

- In manchen Fällen ist es schwieriger, mit den Daten in diesem Format zu arbeiten. Vielleicht bevorzugen wir ein DataFrame mit einer Spalte pro Variable, der durch Zeitstempel in der Datumsspalte indiziert wird.
- Die DataFrame-Methode **`pivot`** führt genau diese Transformation durch:



In [62]:
```
wide_data = long_data.pivot(index='date', columns='variable', values='value')
wide_data.head()
```

Out[62]:

variable	infl	realgdp	unemp
date			
1959-03-31	0.00	2710.349	5.8
1959-06-30	2.34	2778.801	5.1
1959-09-30	2.74	2775.488	5.3
1959-12-31	0.27	2785.204	5.6
1960-03-31	2.31	2847.699	5.2

- Die ersten beiden übergebenen Werte (wenn nicht wie oben explizit angegeben) sind die Spalten, die jeweils als Zeilen- und Spaltenindex verwendet werden, und schliesslich eine optionale Wertespalte, um das DataFrame zu füllen.

`pivot` vs. `unstack`

Beachten Sie, dass `pivot` dem Erstellen eines hierarchischen Indexes mit `set_index` gefolgt von einem Aufruf von `unstack` entspricht:

In [63]:
```
# Zur Erinnerung:
long_data.head()
```

Out[63]:

	date	variable	value
0	1959-03-31	realgdp	2710.349
1	1959-03-31	infl	0.000
2	1959-03-31	unemp	5.800
3	1959-06-30	realgdp	2778.801
4	1959-06-30	infl	2.340

In [64]:
```
# Hierarchischer Index erstellen:
df_HI = long_data.set_index(['date', 'variable'])
df_HI.head()
```

Out[64]:

		value
date	variable	
1959-03-31	**realgdp**	2710.349
	infl	0.000
	unemp	5.800
1959-06-30	**realgdp**	2778.801
	infl	2.340

In [65]:
```
# Nun unstack; gleiches Ergebnis wie mit pivot oben:
df_HI.unstack('variable').head()
```

Out[65]:

	value		
variable	**infl**	**realgdp**	**unemp**
date			
1959-03-31	0.00	2710.349	5.8
1959-06-30	2.34	2778.801	5.1
1959-09-30	2.74	2775.488	5.3
1959-12-31	0.27	2785.204	5.6
1960-03-31	2.31	2847.699	5.2

Fazit

- Wir haben uns nun wichtige Pandas-Grundlagen für den Datenimport, -säuberung und -reorganisation erarbeitet.
- Im nächsten Kapitel werden wir uns mit der *Datenvisualisierung* genauer befassen.
- Wir werden später auf Pandas zurückkommen, wenn wir fortgeschrittenere Analysen (Gruppierungen, Zeitreihen) diskutieren.